

DIVIDE AND CONQUER: BINARY DIVISION

You can carry out binary division, one of the most difficult operations for a computer to perform, in simple μ Ps using low-level instructions.

Performing long division in any number system can be painful. For example, consider how you would divide 14 into 663 using the decimal system. Assuming that you don't have the calculations from Figure 1 in front of you, you would probably think, you can't divide 14 into 6 so you take the next option and divide 14 into 66. But how would you set about doing this division? Most people would probably hunt round in their minds and on their fingers, thinking something like: $3 \times 14 = 42$, but that's too small; $5 \times 14 = 70$, but that's too big; $4 \times 14 = 56$, and that's as close as you can get.

So now you know that the first digit of your result is 4 (from the 4×14). Next, you subtract 56 from 66, leaving 10; drop the 3 down to form 103; and go through the process again: $6 \times 14 = 84$, but that's too small; $8 \times 14 = 112$, but that's too big; $7 \times 14 = 98$, and that's as close as you can get. Thus, you now know that the second digit of your result is 7. Finally, you subtract 98 from 103, leaving 5, which is too small to divide by 14, so you know that your result is 47 with a remainder of 5. (Note that you're performing an integer division.)

A CPU goes through a similar process; it has to perform a series of iterations by taking stabs in the dark, checking to see if it went too far, and backtracking if it did. To further confuse the issue, you end up doing everything somewhat backward, using the same sort of tricks that you employ to make binary multiplications easier for the CPU to handle (Reference 2).

Assume that you wish to divide a 16-bit dividend by a 16-bit divisor (Figure 2). This process is difficult to follow, but everything will come out in the wash. First, you reserve a 2-byte field to store your divisor and a 4-byte field to

This article is an excerpt from the author's forthcoming book, *Bebop Bytes Back* (Reference 1).



CLIVE "Max" MAXFIELD,
INTERGRAPH COMPUTER
SYSTEMS

FIGURE 1

		47	←	RESULT EQUALS 47 PLUS 5 REMAINDER
	14	663		
$4 \times 14 = 56$	→	56		
$66 - 56 = 10$	→	10		
DROP THE 3 TO FORM 103	→	103		
$7 \times 14 = 98$	→	98		
$103 - 98 = 5$	→	5		

Performing long division in decimal requires a little effort.

store your result. Also, you initialize the two most significant bytes of the result to contain all zeros, and you load the two least significant bytes with your dividend (the number to be divided). Once you've initialized everything, you perform the following sequence of operations 16 times:

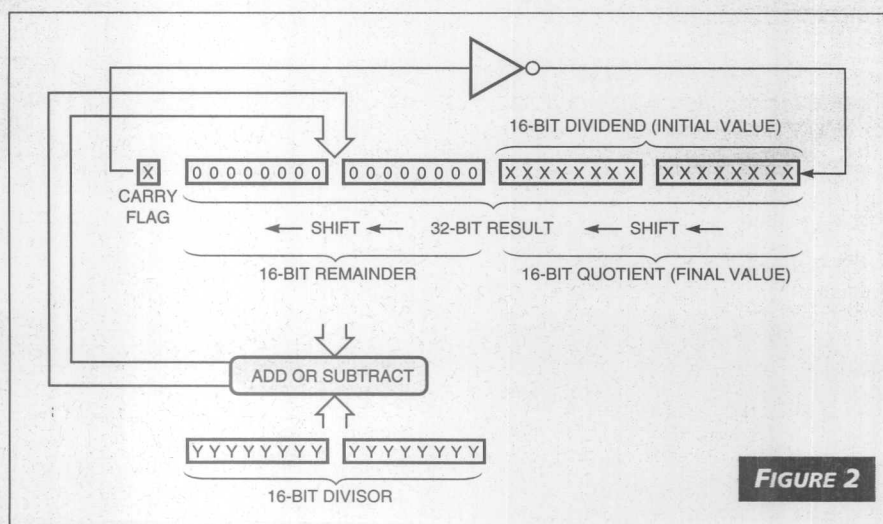


FIGURE 2

Perform unsigned 16-bit divisions using a shift and subtract/add technique.

BINARY DIVISION

1. Shift the 32-bit result 1 bit to the left. (Shift a logic 0 into the least significant bit.)
2. Subtract the 2-byte divisor from the most significant 2 bytes of the result and store the answer back into these 2 bytes.
3. A carry flag with a value of 0 following Step 2 indicates a positive result, which means that the divisor is smaller than the two most significant bytes of the result. In this case, force the least significant bit of the result to 1. A carry flag with a value of 1 indicates a negative result which means that the divisor is bigger than the two most significant bytes of the result. In this case, leave the least significant bit of the result as a 0 (from the shift), add the 2-byte divisor back to the most significant 2 bytes of the result, and store the answer back into these 2 bytes.

Whenever the carry flag contains a 1 entering Step 3, the divisor is too big to subtract from the portion of the dividend that you're currently examining. But you discover this information too late, because you've already performed the subtraction, and you now must add the divisor back in. This process is known as a restoring-division algorithm for just this reason; you have to keep on restoring the divisor every time you "go too far." There are nonrestoring-division algorithms, which are somewhat more efficient, but also a tad more complicated—so just ignore them and hope they'll go away.

Now, unless you have a size 16 brain, these exercises have

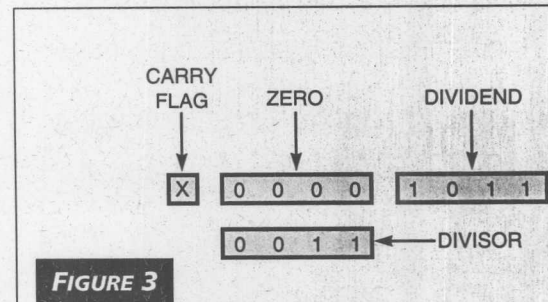


FIGURE 3
Initial conditions provide the starting point for your 4-bit division test case.

probably left you feeling overheated, confused, lost, and alone. Don't be afraid; computer divisions can sometimes bring you to your knees. The easiest way to understand the process is to examine a much simpler test case based on 4-bit numbers. For example, consider how you'd divide 1011 (11 in decimal) by 0011₂ (3 in decimal). The first step is to set up some initial conditions (Figure 3).

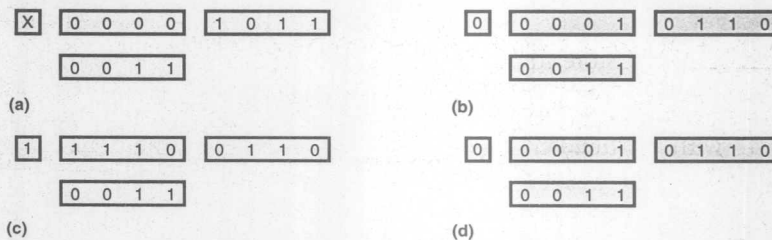
Remembering that this experiment is based on 4-bit numbers, the four most significant bits of what will eventually be your 8-bit result are set to 0, and your dividend will be loaded into the four least significant bits of the result. Now, consider what happens during the first cycle of the process (Figure 4).

Commencing with your initial conditions (Figure 4a), you then shift the entire 8-bit result 1 bit to the left and also shift a logic 0 into the least significant bit (Figure 4b). Next subtract the divisor from the four most significant bits of the result (Figure 4c). However, this step sets the carry flag to a logic 1, which tells you that you've gone too far. Thus, you complete this cycle by adding the divisor back into the four most significant bits of the result (Figure 4d). As fate would have it, the second cycle offers another helping of the same sequence (Figure 5).

Once again, the first thing you do is shift the entire 8-bit result 1 bit to the left and also shift a logic 0 into the least significant bit (Figure 5b). Next, subtract the divisor from the four most significant bits of the result (Figure 5c). This action again sets the carry flag to logic 1, which tells you that you've gone too far. Thus, you complete this cycle by adding the divisor back into the four most significant bits of the result (Figure 5d). You can only hope that the third cycle will do something to break the monotony (Figure 6).

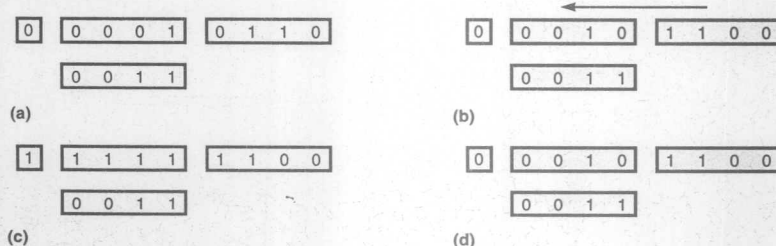
As usual, begin by shifting your 8-bit result 1 bit to the left (Figure 6b) and subtracting the divisor from the four most significant bits of the result (Fig

FIGURE 4



The first cycle of your 4-bit-division test case consists of initial conditions (a), a shift (b), a subtraction (c), and an addition (d).

FIGURE 5



The second cycle of your 4-bit-division test case also consists of initial conditions (a), a shift (b), a subtraction (c), and an addition (d).

ure 6c). In this cycle, however, the carry flag contains a logic 0, so the only thing you have to do is force the least significant bit of the result to a logic 1 (Figure 6d). Now, the excitement really starts to mount, because you've only got one more cycle to go, and you don't appear to be closing in on a result.

In Figure 7, you start by shifting the 8-bit result 1 bit to the left (Figure 7b) and subtracting the divisor from the four most significant bits of the result (Figure 7c). Again, the carry flag contains a logic 0; thus, again, you must force the least significant bit of the result to a logic 1 (Figure 7d).

When you divide 11 by 3, you expect a quotient (result) of 3 and a remainder of 2. The four most significant bits (which represent the remainder) contain 0010_2 (2 in decimal), and the four least significant bits (which represent the quotient) contain 0011_2 (3 in decimal). Good grief, it works!

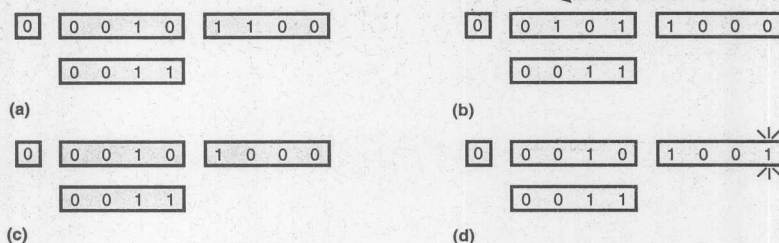
Unfortunately, your division algorithm will not always perform correctly if it encounters negative values, so you have to perform similar tricks to those you used for your signed multiplication subroutines (Reference 2). Thus, you need to check the signs of the numbers first, change any negative values into their positive counterparts using 2's complement techniques, perform the division, and correct the sign of the result if necessary.

Now assume that you're working with an 8-bit μP and you wish to create a subroutine to perform signed binary division on 16-bit numbers. Listing 1 (pg 166) describes just such a subroutine, which retrieves two 16-bit signed numbers from the stack, divides one into the other, and places the 16-bit signed result on the top of the stack. (This subroutine assumes that any 16-bit numbers are stored with the most significant byte "on top" of the least significant byte.)

Note that the 2-byte remainder from the division ends up in the two most significant bytes of the result (`_AH_RES` and `_AH_RES+1`). Thus, if you decide that you want your subroutine to return this remainder, you just add two more pairs of LDA and PUSHA instructions in the `_AH_SAVE` section of the subroutine (just before you push the return address onto the stack). However, usually you create a separate subroutine that returns just the remainder. Alternatively, you could modify this subroutine to have multiple entry and exit points, depending on whether you wish the subroutine to return the remainder or the quotient. Both techniques save you from passing the remainder back and forth when you don't wish to use it.

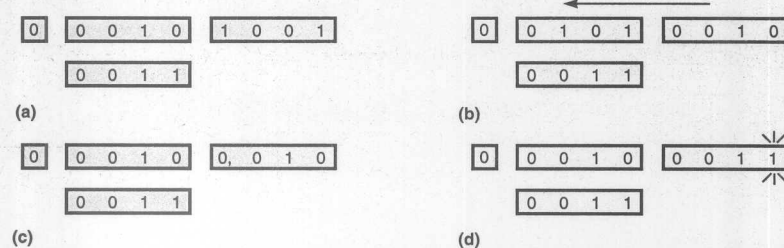
The assembly language in Listing 1 corresponds to no particular μP ; it is designed for the virtual μP implemented in software that accompanies Reference 1.

FIGURE 6



The third cycle of your 4-bit-division test case consists of initial conditions (a), a shift (b), a subtraction (c), and setting the least significant bit of the result to a logic 1 (d).

FIGURE 7



The final cycle of your 4-bit-division test case consists of initial conditions (a), a shift (b), a subtraction (c), and setting the least significant bit of the result to a logic 1 (d).

References

1. Maxfield, Clive "Max," and Alvin Brown, *Bebop Bytes Back (An Unconventional Guide to Computers)*, Doone, Madison, AL. For more details, call 1-800-311-3753 or visit ro.com/~bebopbb/byteback.htm on the Web.
2. Maxfield, Clive "Max," "The product of all fears: binary multiplication," *EDN*, April 10, 1997, pg 195.

Author's biography

Clive "Max" Maxfield is a member of the technical staff at Intergraph Computer Systems (Huntsville, AL), 1-800-763-0242, where he gets to play with the company's high-performance graphics workstations. Maxfield is also the author of *Bebop Bytes Back (An Unconventional Guide to Electronics)* (ISBN 1-878707-22-1). For more information on his forthcoming book, *Bebop Bytes Back (An Unconventional Guide to Computers)*, call 1-800-311-3753. You can reach Maxfield via e-mail at crmaxfie@ingr.com.

Turn the page for Listing 1.

VOTE

Please use the Information Retrieval Service card to rate this article (circle one):

High Interest
594

Medium Interest
595

Low Interest
596

BINARY DIVISION

LISTING 1—16-BIT SIGNED-BINARY-DIVISION SUBROUTINE

```

#####
# Copyright(c) Maxfield & Montrose Interactive Inc., 1996, 1997.
#
# The authors are not responsible for the consequences of
# using this software, no matter how awful, even if they
# arise from defects in it.
#####
# Name: _SDIV16
#
# Function: Divides two 16-bit signed numbers (in the range
#           -32,767 to +32,767): returns a 16-bit signed result
#
# Entry: Top of stack
#         Most-significant byte of return address
#         Least-significant byte of return address
#         MS Byte of 1st 16-bit number (divisor)
#         LS Byte of 1st 16-bit number (divisor)
#         MS Byte of 2nd 16-bit number (Dividend)
#         LS Byte of 2nd 16-bit number (Dividend)
#
# Exit: Top of stack
#        Most-significant byte of result
#        Least-significant byte of result
#
# Modifies: Accumulator
#           Index register
#
# Size: Program = 226 bytes
#        Data = 9 bytes
#####

_SDIV16: BLDX 16 # Load the index register with 16,
                # which equals the number of times
                # we want to go around the loop

        POPA # Retrieve MS byte of return
        STA [_AH_RADD] # address from stack and store it
        POPA # Retrieve LS byte of return
        STA [_AH_RADD+1] # address from stack and store it

        POPA # Retrieve MS byte of the divisor
        STA [_AH_DIV] # from the stack and store it
        POPA # Retrieve LS byte of the divisor
        STA [_AH_DIV+1] # from the stack and store it

# Note that the result is 4 bytes in size (_AH_RES+0, +1, +2,
# and +3), where _AH_RES+0 is the most-significant byte
        POPA # Retrieve MS dividend from stack
        STA [_AH_RES+2] # and store it in byte 2 of result
        POPA # Retrieve LS dividend from stack
        STA [_AH_RES+3] # and store it in byte 3 of result
        LDA 0 # Load the accumulator with 0 and
        STA [_AH_RES] # store it in byte 0 of result then
        STA [_AH_RES+1] # in byte 1 of result

#####
#### Check that we're not trying to divide by zero. If we are then
#### it's an ERROR, so just return zero and bomb out
####
_AH_TSTZ: LDA [_AH_DIV] # Load MS byte of the divisor and OR
        OR [_AH_DIV+1] # it with the LS byte of the divisor
        JNZ [_AH_TSTA] # If the result isn't zero then we've
                        # got at least one logic 1, so jump
                        # to the bit to test for -ve number.
        PUSHA # Otherwise push the zero in ACC onto
        PUSHA # the stack twice, then jump to the
        JMP [_AH_RET] # last chunk of the return routine

#####
#### Invert input values if necessary and load the output flag
####
_AH_TSTA: LDA [_AH_DIV] # Load ACC with MS byte of divisor
        STA [_AH_FLAG] # and save it to the flag
        JNN [_AH_TSTB] # if the divisor is positive then
                        # jump to '_AH_TSTB', otherwise ..

        LDA 0 # ..load the accumulator with 0
        SUB [_AH_DIV+1] # ..subtract LS byte of divisor
        STA [_AH_DIV+1] # ..(no carry in) store result
        LDA 0 # ..load the accumulator with 0
        SUBC [_AH_DIV] # ..subtract MS byte of divisor
        STA [_AH_DIV] # ..(yes carry in) store result

_AH_TSTB: LDA [_AH_FLAG] # Load the flag,
        XOR [_AH_RES+2] # XOR it with MS byte of dividend,
        STA [_AH_FLAG] # then store the flag again

        LDA [_AH_RES+2] # Load MS dividend into the ACC
        JNN [_AH_LOOP] # If dividend is positive then
                        # jump to '_AH_LOOP', otherwise ..

        LDA 0 # ..load the accumulator with 0
        SUB [_AH_RES+3] # ..subtract LS byte of dividend
        STA [_AH_RES+3] # ..(no carry in) store result
        LDA 0 # ..load the accumulator with 0
        SUBC [_AH_RES+2] # ..subtract MS byte of dividend
        STA [_AH_RES+2] # ..(yes carry in) store result

#####
#### Hold tight - this is the start of the main division loop
####
_AH_LOOP: LDA [_AH_RES+3] # Load ACC with LS byte of dividend
        SHL # and shift left 1 bit
        STA [_AH_RES+3] # Store it
        LDA [_AH_RES+2] # Load ACC with MS byte of dividend
        ROLC # and rotate left 1 bit
        STA [_AH_RES+2] # Store it
        LDA [_AH_RES+1] # Load ACC with LS byte of remainder
        ROLC # and rotate left 1 bit
        STA [_AH_RES+1] # Store it
        LDA [_AH_RES] # Load ACC with MS byte of remainder
        ROLC # and rotate left 1 bit
        STA [_AH_RES] # Store it

# Now we want to subtract the 16-bit divisor from the
# most-significant two bytes of the result
        LDA [_AH_RES+1] # Load ACC with LS byte of remainder
        SUB [_AH_DIV+1] # Subtract LS byte of divisor
        STA [_AH_RES+1] # Store it in LS byte of remainder
        LDA [_AH_RES] # Load ACC with MS byte of remainder
        SUBC [_AH_DIV] # Subtract MS byte of divisor (w carry)
        STA [_AH_RES] # Store it in MS byte of remainder

# If the carry flag is zero, set the LS bit of the result
# to logic 1 and jump to the end of the loop. Otherwise
# undo the harm we've just done by adding the 16-bit
# divisor back into the MS two bytes of the result
        JNC [_AH_ADD] # If carry flag not zero jump to
        LDA [_AH_RES+3] # to _AH_ADD, otherwise load ACC with
        OR $01 # LS byte of result, use OR to set LS
        STA [_AH_RES+3] # bit to 1, then store it and jump to
        JMP [_AH_TSTL] # _AH_TSTL (test at end of the loop)

_AH_ADD: LDA [_AH_RES+1] # Load ACC with LS byte of remainder
        ADD [_AH_DIV+1] # Add LS byte of divisor (w/o carry)

```

(continued on pg 168)

BINARY DIVISION

LISTING 1—16-BIT SIGNED-BINARY-DIVISION SUBROUTINE (CONTINUED)

```

STA  [_AH_RES+1] # Store it in LS byte of remainder
LDA  [_AH_RES]   # Load ACC with MS byte of remainder
ADDC [_AH_DIV]   # Add MS byte of divisor (with carry)
STA  [_AH_RES]   # Store it in MS byte of remainder

_AH_TSTL: DECX      # Decrement the index register. If
JNZ  [_AH_LOOP]    # the index register isn't 0 then jump
                        # back to the beginning of the loop

#### Breathe out - this is the end of the main division loop
#### Now check the flag and negate the quotient portion of the result
#### if necessary (see also the notes following the subroutine)
####
_AH_TSTC: LDA  [_AH_FLAG] # Load ACC with the flag
JNN  [_AH_SAVE] # If MS bit of flag is 0 then
                        # jump to '_AH_SAVE', otherwise ..
LDA  0           # ..load the accumulator with 0
SUB  [_AH_RES+3] # ..subtract LS byte of quotient
STA  [_AH_RES+3] # ..(no carry in) store result
LDA  0           # ..load the accumulator with 0
SUBC [_AH_RES+2] # ..subtract MS byte of quotient
STA  [_AH_RES+2] # ..(yes carry in) store result

####
#### Save result on the stack and bug out of here. Remember that
#### we're only returning the 16-bit quotient portion of the result
####

_AH_SAVE: LDA  [_AH_RES+3] # Load ACC with LS byte of quotient
PUSHA                # and stick it on the stack
LDA  [_AH_RES+2] # Load ACC with MS byte of quotient
PUSHA                # and stick it on the stack

_AH_RET: LDA  [_AH_RADD+1] # Load ACC with LS byte of return
                        # address from temp location and
                        # stick it back on the stack
PUSHA
LDA  [_AH_RADD] # Load ACC with MS byte of return
                        # address from temp location and
                        # stick it back on the stack
PUSHA

RTS                # That's it, exit the subroutine

_AH_FLAG: .BYTE    # Reserve 1-byte field to be used as
                        # flag to decide whether or not to
                        # negate the result
_AH_RADD: .2BYTE   # Reserve 2-byte temp location for
                        # the return address
_AH_DIV:  .2BYTE   # Reserve 2-byte temp location for
                        # the divisor
_AH_RES:  .4BYTE   # Reserve 4-byte temp location for
                        # the result. The MS two bytes of
                        # which will contain the remainder
                        # and the LS two bytes the quotient

```

Choose Your Own Path In Life



FLASH 72 PIN SO DIMM

16MB, 12MB, 10MB, 8MB, 6MB, 4MB, 2MB

AVED Memory Products Flash DIMMs allow you to choose the upgrade path that best meets your needs.

- ICs used on the DIMMs are the Intel 28F0xxS5 family of parts
- Smart Voltage Technology™ 2.7V (Read-Only), 3.3V or 5V Vee
 - Flash DIMMs are programmed to run at 85ns
 - Automatic Power Savings Mode



AVED Memory Products
 14192 Chambers Rd. Tustin, CA 92780
 800.778.7928 714.573.5000 714.573.5047 fax
 or call 1-800-573-ASAP for your local All American Sales Representative
 E-mail us at sales@avedmemory.com

CIRCLE NO. 18

168 • EDN MAY 8, 1997

Virtuoso™



"The Best RTOS for DSP"

SHARC

C3x/4x - C62xx

ARM

OakDSPCore

Others...

Eonic Systems Inc.

B-3200 Aarschot Silver Spring, MD 20904
 BELGIUM USA
 Tel. (+32) 16 62 15 85 Tel. (301) 572 5000
 Fax (+32) 16 62 15 84 Fax (301) 572 5005
 e-mail: info@eonic.com e-mail: info@eonic.com

<http://www.eonic.com>

CIRCLE NO. 19